Programming Languages and Compilers
Qualifying Examination

Monday, September 19, 2016

Answer 4 of 6 questions.[1]

GENERAL INSTRUCTIONS

1. Answer each question in a separate book.
2. Indicate on the cover of *each* book the area of the exam, your code number, and the question answered in that book. *Do not write your name on any answer book.*
3. Return all answer books in the folder provided. Additional answer books are available if needed.

POLICY ON MISPRINTS AND AMBIGUITIES

The Exam Committee tries to proofread the exam as carefully as possible. Nevertheless, the exam sometimes contains misprints and ambiguities. If you are convinced that a problem has been stated incorrectly, mention this to the proctor. If necessary, the proctor will contact a representative of the area to resolve problems during the *first hour* of the exam. In any case, you should indicate your interpretation of the problem in your written answer. Your interpretation should be such that the problem is nontrivial.

[1]Note: The full exam had six questions. Participants only answered the five questions included in this document.

Question 1: Temporal Logic

Part (a): CTL

The fundamental operators of CTL are:

- **EX** f, interpreted as "exists next f"
- **EG** f, interpreted as "exists globally f"
- **E**[f **U** g], interpreted as "exists f until g"

Show how each of the following additional CTL operators can be defined in terms of basic Boolean operators $\{\wedge, \vee, \neg, \rightarrow\}$, Boolean constants {true, false}, and the fundamental CTL operators {**EX**, **EG**, **EU**}:

1. **EF** f, interpreted as "exists future f"
2. **AF** f, interpreted as "all future f"
3. **AX** f, interpreted as "all next f"
4. **AG** f, interpreted as "all globally f"
5. **A**[f **U** g], interpreted as "all f until g"

Part (b): Counting in LTL

The fundamental operators of LTL are:

- $\bigcirc\phi$, interpreted as "next ϕ"
- $\Box\phi$, interpreted as "globally ϕ"
- $\Diamond\phi$, interpreted as "future ϕ"
- $\phi\,\mathcal{U}\,\psi$, interpreted as "ϕ until ψ"

Subpart i.

Using only basic Boolean operators $\{\wedge, \vee, \neg, \rightarrow\}$, Boolean constants {true, false}, and the fundamental LTL operators $\{\bigcirc, \Box, \Diamond, \mathcal{U}\}$, write an LTL formula for each of the following properties:

1. ϕ occurs at least twice
2. ϕ occurs at most twice
3. ϕ occurs exactly twice

Subpart ii.

Suppose instead of "twice" in the above properties, we want properties that describe events occurring three times, four times, or in general k times for some fixed k. How do the lengths of the formulae needed to express these properties grow as we systematically increase k?

Question 3: Weakest Preconditions

Consider a simple programming language with the following grammar, where a program P is a sequence S of statements. Note that programs in our language have *no* loops or global variables.

$$\begin{aligned} P &:= S \\ S &:= S; S \\ &\quad \mid x \leftarrow aexp \\ &\quad \mid \text{if } (bexp) \text{ then } S \text{ else } S \end{aligned}$$

where

- x is a program variable, and a program P has a finite set of real-valued variables V. Assume that a non-empty subset $V_I \subseteq V$ is a set of inputs to the program, while all other variables, $V \setminus V_I$, are initialized before use.
- *aexp* is an arithmetic expression over variables V, e.g., $x_1 - x_2 * 3$.
- *bexp* is a Boolean expression over V, e.g., $x > 0$.

Part (a):

Assume we are given a program P with no procedure calls. We would like to prove that P satisfies a *postcondition* φ, which is a formula describing a set of states, where a state is a valuation of all variables V. For example, if φ is $x > 0 \wedge y = 10$, then our goal is to prove that, when P terminates, x holds a positive value and y holds the value 10.

Describe a procedure for proving that P satisfies a given postcondition φ. You may assume that all arithmetic expressions are linear: of the form $a_1 x_1 + \cdots + a_n x_n$, where $a_i \in \mathbb{R}$. Argue that your technique always terminates and produces the right answer.

Hints: Recall that the *weakest precondition* of a statement S with respect to a condition φ, denoted $wp(S, \varphi)$, is the set of states that can reach a state in φ after executing the statement S. For instance, if S is of the form $x \leftarrow E$, then

$$wp(S, \varphi) \equiv \varphi[E/x]$$

That is, all occurrences of x in φ are replaced with E. Assume that you have a weakest precondition oracle for every statement S in the program and formula φ. You may use quantifiers ($\forall$, $\exists$) in your solution.

Part (b):

We now extend program statements as follows:

$$\begin{aligned} S &:= S; S \\ &\quad \mid x \leftarrow aexp \\ &\quad \mid \text{if } (bexp) \text{ then } S \text{ else } S \\ &\quad \mid x \leftarrow proc(x_1, \ldots, x_n) \end{aligned}$$

where *proc* is some procedure for which we have no source code.

Suppose we are given a program P with a *single* procedure call, to some procedure *foo* for which we have no source code. At this point, we cannot really prove that P satisfies some postcondition φ, because we do not know what the call to *foo* returns.

A *procedure summary* of *foo* is a relation $R_{foo}(i_1, \ldots, i_n, o)$ over its inputs $\{i_1, \ldots, i_n\}$ and output o. For example, suppose that *foo* is addition, then $R_{foo}(i_1, i_2, o) \iff o = i_1 + i_2$. In other words, the relation can be described as a Boolean formula over $i_1, \ldots, i_n, o$. If $R_{foo}(i_1, i_2, o) \iff true$, then this means that for any inputs, *foo* can non-deterministically return any output. If for some i_1, i_2 there doesn't exist o such that $R_{foo}(i_1, i_2, o)$, then this means that *foo* does not terminate on inputs i_1, i_2.

Describe a technique that computes the *maximal* relation (the largest one) R_{foo} that ensures that P satisfies the postcondition φ. Argue that there is a unique maximal relation R_{foo} for a given P and φ.

Part (c):

Subpart i.

Provide an example P and φ, where P calls two different procedures, f_1 and f_2, where there is a pair of unique maximal relations R_{f_1} and R_{f_2} that ensure that P satisfies φ. Note that a pair of relations (R_{f_1}, R_{f_2}) is maximal if we cannot enlarge any of them without invalidating the postcondition; they are uniquely maximal if there exists no pair (R'_{f_1}, R'_{f_2}) that is (1) maximal and (2) $R'_{f_1} \neq R_{f_1}$ or $R'_{f_2} \neq R_{f_2}$.

Provide the two relations and argue why they are unique and maximal. The relations should be described as Boolean formulas.

Subpart ii.

Provide an example P and φ, where P calls two different procedures, f_1 and f_2, where there are **NO** unique maximal relations R_{f_1} and R_{f_2} that ensure that P satisfies φ.

Argue why there are no unique maximal relations.

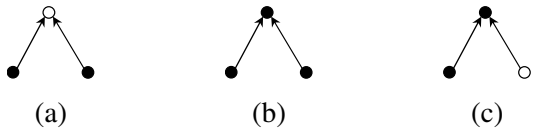


Figure 1: Each of (a), (b) and (c) depicts a small region of the DAG. (a) ⇒ (b): an application of rule 2. (b) ⇒ (c): an application of rule 3.

Question 4: Reverse Execution

This question concerns some ideas related to having an execution environment that can simulate "reverse execution." That is, we wish to consider a run-time environment that provides the illusion that a program can be single-stepped *backward* as well as forward.

Parts (a) and (b) consider an abstract model of a "checkpoint-and-replay" scheme for reverse execution. The model is based on pebbling problems on directed acyclic graphs (explained below). Part (c) concerns a more realistic model of reverse execution.

A *pebbling problem* consists of a directed acyclic graph (DAG), with N nodes and E edges, together with some number of pebbles. The DAG has a distinguished *goal node*, and the objective is to place a pebble on the goal node, subject to the following rules that constrain where a pebble can be placed in a given configuration of pebbles on the nodes of the DAG:

1. A *source* node of the DAG is a node with no predecessors. A pebble can be placed on a source node at any time.

2. A pebble can be placed on a node n when every predecessor of n contains a pebble.

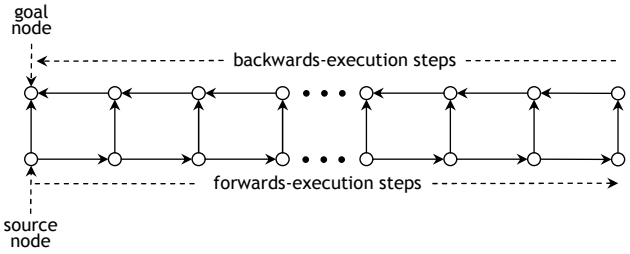
3. A pebble can be removed at any time.

Figure 1 illustrates rules 2 and 3. (Note that rule 1 is a degenerate case of rule 2.)

Note that for an N-node DAG, if you have N pebbles you can obviously pebble the DAG in exactly N rule applications by pebbling the nodes in topological order, and never removing any pebbles. However, when you have fewer than N pebbles (e.g., 3 or $O(\log N)$ or $O(\sqrt{N})$), matters get more interesting: for pebbling problems with fewer than N pebbles, a strategy to pebble the goal node may have to place a pebble on a given node n multiple times (and remove the pebble from n at appropriate moments).

By considering different variants of a pebbling problem—the same family of DAGs, but a different maximum number of allowed pebbles (as a function of N)—one can study time/space trade-offs. One looks at how many rule applications are required to pebble the goal nodes of a family of pebbling problems (where problem instances are characterized by the sizes N and E), as a function of the number of pebbles at your disposal. The maximum number of pebbles required to pebble the goal node provides an abstract measure of *space*; the number of pebbling-rule applications provides an abstract measure of *time* (i.e., the application of each rule takes one time unit).

A crude model of a checkpoint-and-replay scheme for reverse execution is obtained by considering pebbling problems on *ladder graphs*—i.e., pebbling problems on graphs of the form shown below:

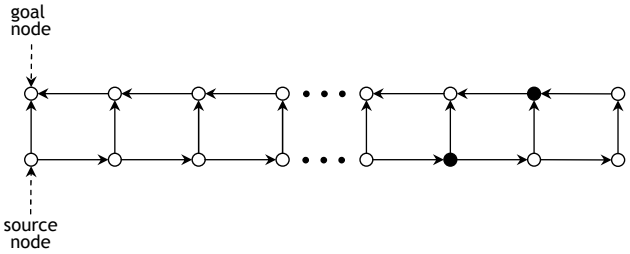


Pebbling the $N/2$ nodes along the bottom row models the steps of *forward execution*: the leftmost node (a source) represents the initial state S_1; the node to its right represents the state S_2 after one step of execution; etc. (Each pebble on the bottom row, other than the rightmost one, represents a checkpoint.)

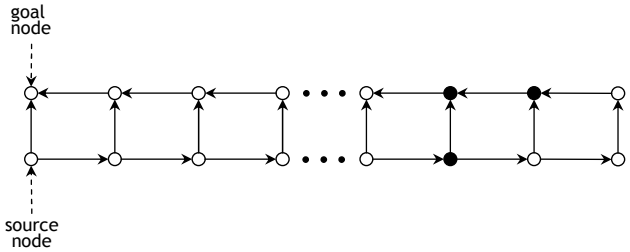
Pebbling the $N/2$ nodes along the top row models the outcome of *backward execution* steps being presented to the user. The placement of a pebble on the top-rightmost node represents the run-time environment presenting to the user state $S_{N/2}$, which is the first state of the backward-execution sequence; the placement of a pebble on the node to its immediate left represents the run-time environment presenting to the user the second state in the backward-execution sequence (state $S_{N/2-1}$); etc.

A ladder-graph pebbling problem of size N models passing through $N/2$ successive states of forward execution, followed by backward execution through the same $N/2$ states. Moreover, ladder-graph pebbling models a rather crude model of reverse execution (compared to the one that will be considered in Part (c), for instance) where each state is considered to be a monolithic whole. That is, one cannot create S_{i-1} from S_i just by changing S_i in some way; instead, S_{i-1} can only be created via replay—i.e., by *forward* execution either from some checkpoint (represented by a pebble in the bottom row further to the left) or from the beginning (starting over from the source node in the lower-left-hand corner).

If the user has been presented with S_i (modeled by a pebble on the i^{th} node from the left in the top row), and the run-time environment has re-created S_{i-1} by some amount of forward execution from a checkpoint (modeled by a pebble on the $(i-1)^{\text{st}}$ node from the left in the bottom row), S_{i-1} can be presented to the user (modeled by the placement of a pebble on the $(i-1)^{\text{st}}$ node from the left in the top row). For instance, the following diagram represents the situation in which $S_{N/2-1}$ has been presented to the user, and $S_{N/2-2}$ has been re-created by some amount of forward execution.



The following diagram represents the presentation of $S_{N/2-2}$ to the user as the third step of the backward-execution sequence.



Part (a):

Explain how with 3 pebbles the goal node of a ladder-graph pebbling problem can be pebbled using $O(N^2)$ rule applications.

Part (b):

Answer **one** of the following two questions:

1. Explain how with $O(\log N)$ pebbles the goal node of a ladder-graph pebbling problem can be pebbled using $O(N \log N)$ rule applications. Hint: use divide-and-conquer.

2. Explain how with $O(\sqrt{N})$ pebbles the goal node of a ladder-graph pebbling problem can be pebbled using $O(N)$ rule applications. Hint: Set up and solve in succession $O(\sqrt{N})$ subproblems that can each be solved with $O(\sqrt{N})$ pebbles using $O(\sqrt{N})$ rule applications.

Part (c):

Suppose that instead of a checkpoint-and-replay scheme as considered in Parts (a) and (b), we wish to use a "logging" scheme in which the *changes* made by each execution step are logged. (Assume that the programming language is a simple imperative programming language with integer variables and procedure calls.)

1. Explain what actions the run-time environment would have to perform during forward execution to identify and log changed portions of the state. Describe the data structure (or structures) that the run-time environment could use to log changes.

2. Explain what actions the run-time environment would have to perform during simulated backward execution.

Question 5: Pointer Analysis

This question concerns flow-insensitive pointer analysis in a language with the following kinds of assignment statements:

```
   ID = &ID
   ID = ID
   ID = *ID
  *ID = ID
```

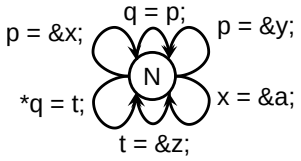
The goal of the analysis is to discover, for each variable `x`, which variables `x` might point to. Because we are interested in a flow-*in*sensitive analysis, we ignore the actual control flow in the program being analyzed; that is, we treat the program as if there were an edge in its control-flow graph from every node to every other node (so the statements could be executed in arbitrary order, and each statement could be executed an arbitrary number of times). We can accomplish this using a "flow-insensitive" control-flow graph that has just a single node N as well as an edge $N \xrightarrow{a} N$ for each edge (with associated action a) in the original control-flow graph.

For example, given below are the statements of a program and the desired result of performing flow-insensitive pointer analysis on the program.

Program	Result of Pointer Analysis

`p = &x;`
`q = p;`
`p = &y;`
`x = &a;`
`t = &z;`
`*q = t;`
	`p` might point to `x` or `y`
	`q` might point to `x` or `y`
	`x` might point to `a` or `z`
	`y` might point to `z`
	`t` might point to `z`

where the program's "flow-insensitive" CFG is



Remember, the fact that "`p = &y`" is shown after "`q = p`" in column 1 of the table above is irrelevant—the statements are treated as if they could execute in any order, as illustrated by the CFG.

Part (a):

One way to define our desired pointer analysis is using a standard dataflow analysis applied to the "flow-insensitive" control-flow graph. Recall that, in general, one way to define a standard dataflow analysis is by supplying

1. A lattice of values (where each value represents one "dataflow fact") including a join operator (or meet operator, depending on how you orient the lattice).

2. For each CFG edge $e_{m,n} = m \rightarrow n$, there is a dataflow function $f_{m,n}$ that maps the dataflow value $val(m)$ that holds at m to an outgoing dataflow value, $f_{m,n}(val(m))$, and the dataflow value that holds at n is
$$\bigsqcup_{m} f_{m,n}(val(m)).$$

Define a standard dataflow analysis so that for the unique node N of the flow-insensitive CFG, the final value obtained for $val(N)$ is the desired map from variables to their points-to sets.

Part (b):

Another way to define our pointer analysis is using Horn clauses. The idea is that each assignment statement in the program generates one Horn-clause fact, and that there are a set of Horn-clause rules that define how to infer *pointsTo*(_, _) facts from other facts.

For example, the assignment "`p = &x`" in the program would cause the fact "$assignAddr(p, x)$." to be created, and there would be a rule

$$pointsTo(A, Z) :- assignAddr(A, Z).$$

From the above rule and the fact "$assignAddr(p, x)$.," we would obtain the fact "$pointsTo(p, x)$."

Similarly, the assignment "`q = p`" in the program would cause the fact "$assign(q, p)$." to be created. Moreover, there would be a rule

$$pointsTo(Q, X) :- assign(Q, P), pointsTo(P, X).$$

Note that we can use the two facts "$assign(q, p)$." and "$pointsTo(p, x)$." and the rule above to obtain the fact "$pointsTo(q, x)$." In general, one can read a rule from right to left: the right-hand side indicates a pattern of zero or more facts in the database of facts; the left-hand side indicates a possibly new fact to be added to the database of facts.

You are to finish the definition of the pointer analysis via Horn clauses by (i) stating what kind of Horn-clause fact should exist for each of the remaining two kinds of assignment statements, and (ii) defining two more Horn-clause rules that permit the deduction of the appropriate additional pointsTo facts from those rules.

Part (c):

Note that the desired set of *pointsTo* facts that are to be obtained using this technique on the program shown above is:

$pointsTo(p, x).$
$pointsTo(p, y).$
$pointsTo(q, x).$
$pointsTo(q, y).$
$pointsTo(x, a).$
$pointsTo(x, z).$
$pointsTo(y, z).$
$pointsTo(t, z).$

Illustrate your Horn-clause facts and rules by giving the initial set of facts that would be generated for the example program, and then showing how to use those facts and the four Horn-clause rules of the pointer-analysis method to determine the eight *pointsTo* facts listed above.

Question 6: Adding Generators to C

We would like to add a new feature, *generator functions*, to the C language. Standard functions produce a single result value, but generator functions produce a sequence of values. The caller of a generator function can consume these values one at a time until the generator function finally runs out of values to produce.

We need a new syntax for producing a sequence of values instead of a single value. A generator function uses the "**yield** *e*" statement to produce one value *e* in a sequence. However, a generator function does not stop after a **yield** in the way that standard functions stop after `return`. Instead, the generator function continues executing at the statement following the **yield**. If it reaches additional **yield** statements, then it produces additional values. For example, a **yield** in a loop may produce a long sequence of values until the loop itself finally terminates. A generated sequence ends when the generator function executes `return` with no arguments or simply falls off the end of the function body. A generator function can also end using "`return` *e*" if *e* itself evaluates to a generated sequence of appropriate type; this behaves as though all the elements of *e* were added after any elements already produced.

We introduce a new type, "τ **sequence**", to represent the type of generated sequences of τ elements. A generator function is a function that returns a sequence type. For example, the following defines `primaries` as a generator function that returns a sequence of names of primary colors:

```
char * sequence primaries() {
  yield "red";
  yield "green";
  yield "blue";
}
```

We need one more new piece of syntax: if *s* is a generated sequence, then "**next** *s*" is an expression that evaluates to a pointer to the next value in sequence *s*, and as a side effect advances *s* forward one position. Thus, one can examine the entire sequence *s* by repeatedly evaluating "**next** *s*". We choose to have **next** return a pointer to the next value so that we can easily mark the end of a stream: when *s* has no more values, "**next** *s*" returns `NULL`. For example, the following defines a generator of integer ranges and uses it to print a list of numbers:

```
int sequence range(int low, int high) {
  int current;
  for (current = low; current < high; ++current)
    yield current;
}

void tester() {
  int sequence values = range(1, 20);
  for (const int *num = next values; num != NULL; num = next values)
    printf("next value is %d\n", *num);
}
```

Part (a): Writing Generator Functions for Iteration

Suppose you have binary search trees of integers with nodes sorted in ascending order. Each node represented by the following structure:

```
struct Node {
  int value;
  struct Node *left;
  struct Node *right;
};
```

Write a generator function that, given a binary search tree, produces the values from the tree in descending (reverse) order. Your function should take a pointer to the tree's root node as its only argument and should produce a descending-order `int` **`sequence`** result.

Part (b): Type Checking

Give type checking rules for **`yield`** statements and **`next`** expressions. Your answer should be in the form of formal rules of inference and judgments on a typing environment, as in these examples for `if` statements, `return` statements, integer addition:

$$\frac{\Gamma \vdash e : \tau \quad \tau \text{ is a scalar type} \quad \Gamma \vdash b_1 : \texttt{void} \quad \Gamma \vdash b_2 : \texttt{void}}{\Gamma \vdash \texttt{if (}e\texttt{)}\ b_1\ \texttt{else}\ b_2 : \texttt{void}}$$

$$\frac{\Gamma \text{ indicates that the function currently being checked has return type } \tau \quad \Gamma \vdash e : \tau}{\Gamma \vdash \texttt{return}\ e\texttt{;} : \texttt{void}}$$

$$\frac{\Gamma \vdash e_1 : \texttt{int} \quad \Gamma \vdash e_2 : \texttt{int}}{\Gamma \vdash e_1\ \texttt{+}\ e_2 : \texttt{int}}$$

Notice that we assume that Γ is capable of recording the declared return type of the function currently being checked, if any. This would require supporting treatment in the type checking rule for functions, which we have not included here.

Briefly describe your assumptions about any information available in typing environments that is required in order to type check the two new language features. If you introduce any new symbols, notation, supporting functions, or other infrastructure, explain what you add, what it means, and why it is necessary.

Part (c): Implementation Choices

There are several possible implementation strategies one could use for generator functions. These include:

all-at-once: The entire generated sequence is built at once when a generator function is called, and **`next`** simply iterates over the stored sequence.

on-demand: The generator function is not run at all until **`next`** is actually used to demand a value. Each time **`next`** is evaluated, the generator function runs to the next **`yield`**. The yielded value is the value for **`next`**, and the generator function is suspended after the **`yield`** until the next time another value is needed.

concurrent: The generator function is run in a separate thread, and executes concurrently with the rest of the program. The generator function's **`yield`** statements and the caller's **`next`** evauations are in a producer/consumer relationship. They are connected using a fixed-size buffer of n sequence elements that have been produced but not yet consumed, where n is implementation-defined.

Subpart i.

Briefly discuss advantages and disadvantages of these three strategies. You may take into consideration language design, ease of implementation, or any other practical concern that would be important when creating a real language. Assume a realistic runtime environment, with limited memory and processors that run at finite speeds.

Subpart ii.

Write a small program that prints "`all-at-once`" or "`on-demand`" depending on which behavior is being used. You do not need to detect the concurrent style of behavior. Assume that memory is limitless and processors are infinitely fast.

Subpart iii.

In a purely functional program with no side effects, which of the three semantic styles above can be distinguished from each other, and which must always give identical behavior for all programs? Assume that memory is limitless and processors are infinitely fast.

Part (d): Implementation Details

Propose a concrete runtime implementation for generator functions using on-demand semantics. Focus your description on the following:

- Does a variable of type τ **`sequence`** occupy storage space at runtime? If so, how much space and what information is stored there?
- What happens at runtime when a generator function is called?
- What happens at runtime when "**`next`** *s*" is evaluated?

Assume that our starting point is a typical C runtime environment with a stack, a heap, stack frames, a program counter, etc. Your solution must not require multiple hardware threads.

We have several design goals here, and it may not be possible to satisfy all of them completely. We would like generators to have good performance, both in terms of speed and in terms of memory. For reasons of stability, we would also like to avoid changing the runtime environment any more

than necessary. In particular, parts of a program that do not use generators should continue to behave very much like they did in standard C. Lastly, we would like to avoid arbitrary limits on how generators can be written: generators should be able to call other generators, call themselves recursively, call standard functions, and so on.

If your solution favors some of these goals by sacrificing others, explain the trade-offs and justify why the design you chose represents a good compromise.